Preserving Architectural Decisions through Architectural Patterns

Minh Tu Thon That, Salah Sadou, F Oquendo, R Fleurquin

Noname manuscript No.
(will be inserted by the editor)

Preserving Architectural Decisions through Architectural Patterns

Minh Tu Ton That · Salah Sadou · Flavio Oquendo · Régis Fleurquin

Received: date / Accepted: date

Abstract Architectural decisions have emerged as a means to maintain the quality of the architecture during its evolution. One of the most important decisions made by architects are those about the design approach such as the use of patterns or styles in the architecture. The structural nature of this type of decisions give them the potential to be controlled systematically. In the literature, there are some works on the automation of architectural decision violation checking. In this paper we show that these works do not allow to detect all possible architectural decision violations. To solve this problem we propose an approach which: i) describes architectural patterns that hold the architectural decision definition, ii) integrates architectural decisions into an architectural model and, iii) automates the architectural decision conformance checking. The approach is implemented using Eclipse Modeling Framework and its accompanying technologies. Starting from well-known architectural patterns, we show that we can formalize all those related to the structural aspect. Through two evaluations, we show that our approach can be adapted to different architecture paradigms and allows to detect more violations comparing to the existing approaches.

Keywords Architectural decision · Pattern · Model Driving Engineering

M.T. Ton That
Universite de Bretagne Sud, IRISA, Vannes, France
E-mail: minh-tu.ton-that@irisa.fr

S. Sadou
Universite de Bretagne Sud, IRISA, Vannes, France
E-mail: Salah.Sadou@irisa.fr

F. Oquendo
Universite de Bretagne Sud, IRISA, Vannes, France
E-mail: Flavio.Oquendo@irisa.fr

R. Fleurquin
Universite de Bretagne Sud, IRISA, Vannes, France
E-mail: regis.fleurquin@irisa.fr

1 Introduction

Software architectures deal with the decomposition of a system into its major components, the mechanisms and rules by which these components interact and the global properties of the system that emerge from the composition of its pieces. Documenting software architectures during the development produces major benefits such as early analysis, system visibility and complexity management, design discipline and global conceptual integrity management. During the maintenance, architecture models can help software developers to gain a sufficient knowledge of the software. Thus, these high level models play now a central role in all the stages of the software processes. But to be actually useful such models cannot be confined just to a simple "blueprint" role. They should also provide knowledge about the "reasons" that led to the adoption of this particular architecture. An architecture model should help in pointing and understanding the usage of a particular component, organization, style or pattern. A good software architecture must talk about itself. This self-content attitude is very important particularly during the maintenance. Without it, any evolution of the system becomes potentially time-consuming and risky. Indeed, a change maybe in contradiction with some previously taken decisions, which lead the system to lose some of its properties (such as maintainability, portability or performance). When the problem is detected, a rework is needed. It is a sequence of iterations, that undoubtedly increases the development costs. To avoid this problem, we should i) make explicit this knowledge and ii) check (automatically if possible) that a change to be made does not conflict with past decisions.

Usually, the documentation of this knowledge takes the form of a set of what we call an *architectural decision* (AD) [17]. ADs cover many aspects of an architecture. Among these aspects, those related to the structure of the architecture have the highest potential to be automatically checked. This type of AD highlights a particular subset of an architecture model (a subset of the model elements) and conveys its *rationale*. For instance, a structural AD can stipulate that a pipeline style is introduced in some part of an architecture to ensure a certain level of maintainability for a specific function. Because the capitalization of structural ADs is a central concern, several proposals have already been made to formally document and check them. Most of these proposals rely on the concept of *pattern*. This choice is due to the fact that at the architecture level, a typical design decision is about applying some architectural patterns or styles that address the quality requirements of the system [3,41]. The use of architectural patterns as structural solutions of ADs (called StAD in the remaining of this paper) helps the architect to have a general, high level view about an architecture [43,45].

There are mainly two trends of proposal in the literature concerning the automatic checking of StAD. The first trend such as [38,39] focuses on the respect of the applied pattern in the architecture model. StADs take the form of architectural constraints describing the organisation imposed by the pattern. These constraints are carried by the impacted model elements. Modifications

or additions to the model that cause a violation of the structure (constraints) of a pattern can be detected. The drawback of using these approaches to document StADs is that the consistency is only checked as long as concerned elements structurally conform to their playing roles in the architecture. Consequently, the existence of pattern-related elements is a condition that is left unchecked which can lead to undetected violations when deletions are done. On the contrary to these mentioned work, the second trend consists of several studies such as [18,19,8] which are conducted to define the trace of applying patterns in the architecture. Instead of maintaining the structural consistency of a pattern, these works focus on verifying the existence of the impacted model elements. Deletions of these elements are detected but they can cause, as modifications of these elements, some false positive. Moreover additions that would affect the consistency of a pattern are not detected. It is unsure that the semantics of applied patterns' structure is assured during the architecture's evolution. Thus, none of these proposals can ensure that all the StAD violations will be detected.

In this paper we propose an additional support to the documentation of StADs which helps architects to automatically preserve architectural knowledge about the structure of architecture. We chose also to use architectural patterns as a support for describing StADs. We make explicit the links between the pattern's elements and the architecture's elements through a mapping. Thanks to this mapping, after an evolution, it becomes possible to discover that a StAD is no longer implemented in the architecture, even if all the involved elements from the architecture were removed. Thus, our approach emphasizes the usefulness of combining mapping models and pattern models in documenting StADs. Indeed, mapping models and pattern models together maintain the existence of the StAD and its structural consistency. The absence of one of these two artefacts will lead to an incomplete StAD and thus, undetected violations.

The remaining of the paper is organized as follows. Section 2 introduces the vocabulary and concepts related to ADs and patterns. Section 3 gives an illustration of the problem and shows the limitations of the existing approaches. Section 4 introduces the general approach while Section 5 goes into detail of the reusable StAD creation step. Section 6 describes StAD manipulation stage. Section 7 introduces the implemented toolset and Section 8 two validations we have conducted. Before concluding the paper in Section 10, we discuss related work in Section 9.

2 Background

We conducted a review of the relevant literature about AD documentation and pattern modeling and how the latter can be used to support the former. More specifically, we concentrated on ADs about the use of patterns in the architecture. We discuss these issues in the sub-sections below.

2.1 Architectural Decision (AD)

An AD is defined as the description of a set of any modification made to the software architecture together with its rationale, design rules and design constraints [17]. AD is structured by ISO/IEC/IEEE 42010 [16] as a concept that affects *architectural description elements*, pertains to one or more *concerns* and justifies *architectural rationale*. An *architectural description element* could be a stakeholder, a viewpoint, a model element, etc. A *concern* could be any interest in the system such as behavior, cost, structure, etc. An *architecture rationale* is the explanation, justification or reasoning about the architecture decisions that have been made and architectural alternatives not chosen. This definition comes after many efforts in the literature to draw a complete structure for AD such as [17, 40, 46].

AD documentation serves as a means to emphasize the rationale behind some design decisions having been made. Respecting these rationale means that the architecture evolves in harmony with existing design decisions. Thus, architecture is the result of making a set of ADs, and by documentation the outcomes of those ADs are recorded [9].

2.2 Pattern

A pattern presents a well-proven solution for a particular recurring design problem that arises in a specific design context [6]. Patterns serve many different purposes. They provide common vocabulary and understanding for design principles. Patterns themselves are a means of documenting software architecture. But most importantly, patterns support the construction of software with well-defined properties [12]. That is the reason why a typical method of software architecture development is to select and combine a number of patterns that address the expected quality requirements and use them to build elements of the architecture [3, 5, 41, 35]. Therefore, pattern-centric software architectures are built around the notion of pattern. Such a software architecture structure can be seen to be an amalgamation of many different patterns.

2.3 StAD about the application of patterns

ADs can fall into many different categories: ADs about the applied technology, the system assets or the conceptual model of the architecture [46]. Among them, structural ADs are the most common ones [30]. Indeed, every evolution of a system often begins with structural ADs taken to modify the structure of architecture since they affect the system on the highest level of abstraction. StAD is a part of structural AD. StAD highlights certain structures in the architecture and links them to structural ADs [31]. It not only conveys the intention of architects but also shapes the structure of the architecture. Thus, a StAD-conformed structure within the architecture not only represents the

intention of an AD but also, in terms of AK (architectural knowledge), is the indicator of the existence of AK. One of the most popular StADs are those concerning the application of patterns in the architecture. Indeed, patterns concern some of the most important AD and provide a rich set of AK [43]. Moreover, reusable design knowledge normally documented in patterns can be adapted to inexpensively document AD in specific context [47]. Thus, StADs about the application of architectural patterns become important artefacts that need to be documented. Applying patterns means applying successive StADs that eventually result in an architecture [9]. In other words, StAD claims the reason for which applied patterns prevail in the structure of architecture.

2.4 Pattern Conformance

Patterns, at the architectural level particularly, have become important constructs in existing ADLs such as Wright [1], Acme [13] or UML [11]. Patterns allow one to define a domain-specific design vocabulary, together with constraints on how that vocabulary can be used in constructing an architecture. In these ADLs, patterns are initialized by declaring instances of architectural elements and typing them with pattern elements. An architecture is said to conform to a pattern if there is no conflict between pattern-typed architectural elements with respect to pattern constraints. In Acme and Wright, constraints are written based on first-order predicate logic language and pattern consistency is verified by formal specification checkers [1, 13]. In UML, constraints, also known as well-formed rules, are written using OCL (Object Constraint Language)[28] and pattern consistency is referred to the conformance of model against meta-model. For instance, in [11] the authors specify pattern by meta-models which in turn, characterize UML design models. In another work [24], the authors use UML's stereotype extension mechanism to incorporate other ADL's features, including the description of architectural styles, which are used to verify the conformity of UML models.

2.5 StAD Conformance

In the structural point of view (skipping informal information such as the context, the problem, the rationale, etc.), a StAD is thus considered as any addition, subtraction, modification made to the structure of software architecture. An architecture is said to conform to a StAD if all of StAD-related elements prevail in the architecture [18, 19]. This understanding of StAD conformance implies two requirements: i) given an architectural element, one should be able to trace back to the architectural decision which it is based on and ii) the main consequences of an executed StAD, or the changed elements in the model due to that StAD in other words, must be preserved in the architectural model. In case of StAD about the application of pattern, StAD-related

elements are in fact those playing roles in the applied pattern. Thus, the StAD conformance implies that the outcome of the application of that pattern must be documented.

3 Illustration and problem

To illustrate the need of maintaining StADs and how existing work cannot respond to this need, we take the FRC (Forestry Regulatory Commission) case study which is described in [36]. FRC is dedicated to commercial activities related to the forestry industry. It has been expanded in the past and expects to continue to change which results in costly development. Therefore, SOA (Service Oriented Architecture) has been opted to help the system respond more quickly to changing requirements. During the transition step to SOA solution, instead of rebuilding existing components, it was considered faster and less expensive to reuse them. It is the case of the *Fines service* and the *Evaluations service* which want to access to different repositories managed by a legacy component called *Data Controller*, a standalone Java EJB (Enterprise Java Beans). The *Legacy Wrapper* pattern [36] was chosen to handle this situation as illustrated in Figure 1.

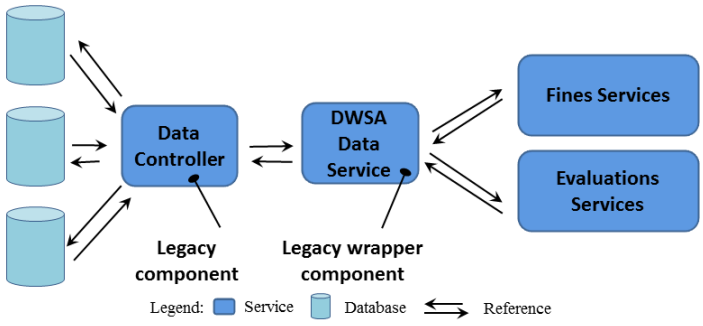

Fig. 1 Legacy Wrapper pattern in FRC

More specifically, the *DWSA Data Service* which is deployed as a web-service is added to wrap the legacy component *Data Controller* to assure a seamless communication. The advantage of this pattern is that it allows the *Data Controller* component to perform changes and refactoring efforts without affecting the other service consumers that bind to it. This pattern implies the constraint stipulating that every service can only connect to the *Data Controller* component via the wrapper *DWSA Data Service*. The application of the Legacy Wrapper pattern in the FRC architecture is a StAD that needs to be taken into consideration throughout the evolution of FRC. In the viewpoint of the existence of related elements, this StAD is considered to be preserved as long as the legacy component *Data Controller* and the wrapper *DWSA Data*

Service exist in the architecture of FRC. On the other hand, in the viewpoint of the pattern's structural consistency, this StAD is maintained as long as the legacy component *Data Controller* is the only component being able to access to the wrapper *DWSA Data Service*.

Later, a new service called *Appealed Assessments* was added to FRC and it also needs to access the *Data Controller* component. The architect that decided this addition was not completely aware of the rationale behind the existence of the wrapper *DWSA Data Service*. He then decided to use the Service Façade pattern [36]. More specifically, a façade called Data Relayer is added inside the *Appealed Assessments Service* with the only purpose to communicate with the component *Data Controller* (Figure 2). The reason influencing the architect not to use the Legacy Wrapper pattern is that Service Façade is simpler to implement, although the service using a façade will be coupled to the legacy component.

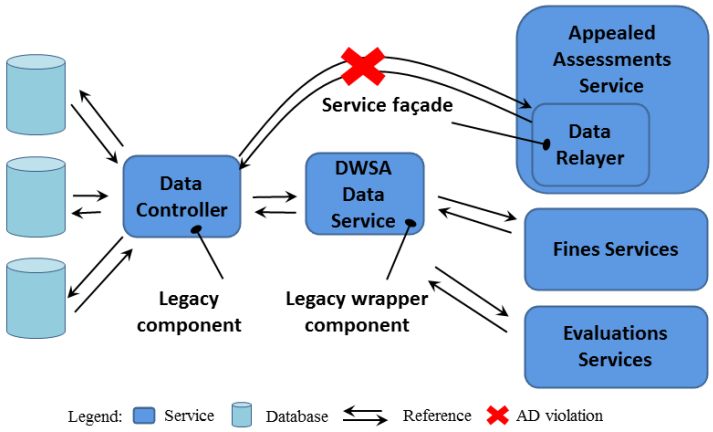

Fig. 2 Architectural decision violation by adding Service Façade pattern

As we can observe in Figure 2, the constraint of the *Legacy Wrapper* pattern is violated due to the fact that the Appealed Assessments Service can still connect to the *Data Controller* component via the *Data Relayer* façade without passing through the wrapper *DWSA Data Service*. In spite of the existence of the legacy component *Data Controller* and the wrapper *DWSA Data Service*, the decision of using Legacy Wrapper pattern has been violated since its structural consistency is not insured. This example shows that the existence of StAD-related elements is a necessary condition but not a sufficient one to detect the violation of StADs. Indeed, one indispensable part of architectural pattern is constraints imposed on future evolution of concerned elements.

Once informed about the violation, the architect changed the Legacy Wrapper pattern to a less rigid version which allows façade components to connect to legacy components. Thus, the link between *Appealed Assessments Service* and *Data Controller* is no longer a violation. Later, another architect par-

ticipated in the project. He was not aware of the extended version of the Legacy Wrapper and he found that the *Appealed Assessments Service* must not access the *Data Controller* component directly. He deleted the link from the *Appealed Assessments Service* component towards the legacy component *Data Controller* and the *Data Relayer* façade as well. Then he added a link between the *Appealed Assessments Service* component and the legacy wrapper *DWSA Data Service*.

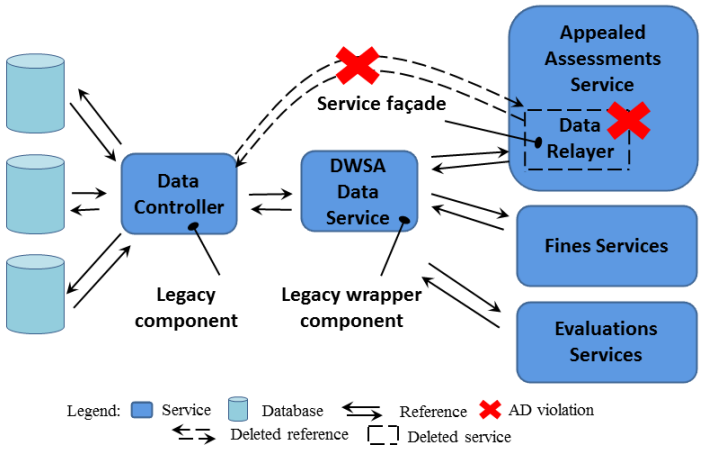

Fig. 3 Architectural decision violation by deletion

Despite that the structural consistency of Legacy Wrapper pattern is always insured (there is no direct access to the legacy component), the decision of allowing the *Data Relayer* component (façade) to access directly to the *Data Controller* component (legacy component) as an extended version of Legacy Wrapper pattern has not been preserved as shown in Figure 3. Thus, the structural consistency of pattern is also a necessary condition but not a sufficient one to detect the violation of StADs. Indeed, the obvious prerequisite of an architecture conformed to a given pattern is that the concerned elements must exist.

In summary, these examples show that the documentation of StADs about the application of patterns should focus on two aspects: the existence of related elements and the structural consistency of the applied patterns. Moreover, they are complementary aspects and both of them must be considered in evaluating StADs. In other words, one aspect can not replace the role of the other one and vice versa. The lack of one of these two aspects could lead to undetected StAD violations. In the literature, the existing work about ADLs [1,13] or architectural constraints [37,38] focus solely on the structural consistency aspect. The validity of StADs is maintained as long as concerned elements structurally conform to their playing roles in the architecture. Whereas, the other work about ADs [19,21,18] on the other hand, focus on the existence aspect. A

StAD is considered to be preserved as long as modifications to concerned elements persist in the architecture. The approach presented in the remaining of this paper consists in combining these two aspects in documenting StADs.

4 General Approach

The main idea behind our work is that StADs about the use of patterns should be preserved throughout the evolution of the architecture. More specifically, the existence of pattern-related elements and the structural consistency of StADs about pattern use should be automatically checked whenever there is a modification to the architectural model. Because we only concentrate on StADs about pattern use and for the sake of simplicity, *the term StAD throughout the following this paper is understood as StAD about pattern use.* Moreover, we focus on the structural part of AD to support the conformance checking. It does not mean that the other parts of AD such as the rationale or the concerns are not important. Instead, together they make a complete structure of StADs that supports both the documentation and the automatic checking.

Similar to [46], StAD documentation in our approach goes through three steps: Pattern creation, StAD integration and StAD verification. Figure 4 depicts the process of using StADs in architecture construction.

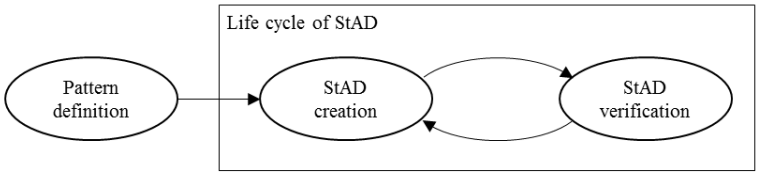

Fig. 4 The process of using StAD

Pattern creation consists in the specification of a pattern structure. A pattern is defined once and used for all concerned StADs. *StAD creation* is the step in which the decision about the application of the defined pattern is created. Finally during the *StAD verification* step, the architectural model is checked whether it complies with the created StAD. If the architectural model is found to be inconsistent with the created StAD, the architect can come back to the *StAD creation* step and recreate another StAD and so forth.

On the purpose of automating the process of StAD documentation, we use the Model Driven Architecture (MDA) approach [27]. Each artifact is considered as a model conforming to its meta-model in order to create a systematic process thanks to model transformations and leverage existing MDA techniques (e.g. conformity verification).

In the remainder of this section, we will go further into each step in the StAD documentation process.

4.1 Pattern definition

We propose the use of a *general pattern language* for the purpose of pattern definition. As shown in Figure 5 (Pattern definition part), the abstract syntax of this language is described using a *general pattern meta-model* which contains only architectural elements involved in the pattern definition. These elements are determined through a survey of well-known architectural patterns such as those described in [36,9,6]. In terms of concrete syntax of our language, one can graphically define a meaningful architectural pattern in form of a *pattern model* using necessary elements. Furthermore, *pattern models* are also language-independent. With the separation between pattern definition and architectural design, no modification to the architectural model is needed to define a pattern, which makes it easy to adapt to different ADLs.

4.2 StAD creation

Links between pattern elements and their correspondent architectural elements play an important role in keeping track of StAD made to an architectural model. An explicit linking will facilitate the specification of StADs as well as their storage. In our approach, links between pattern elements and architectural elements are represented by *mapping models* (illustrated in the Pattern integration part of Figure 5). A *mapping model* indicates that a StAD has been applied on an architectural model.

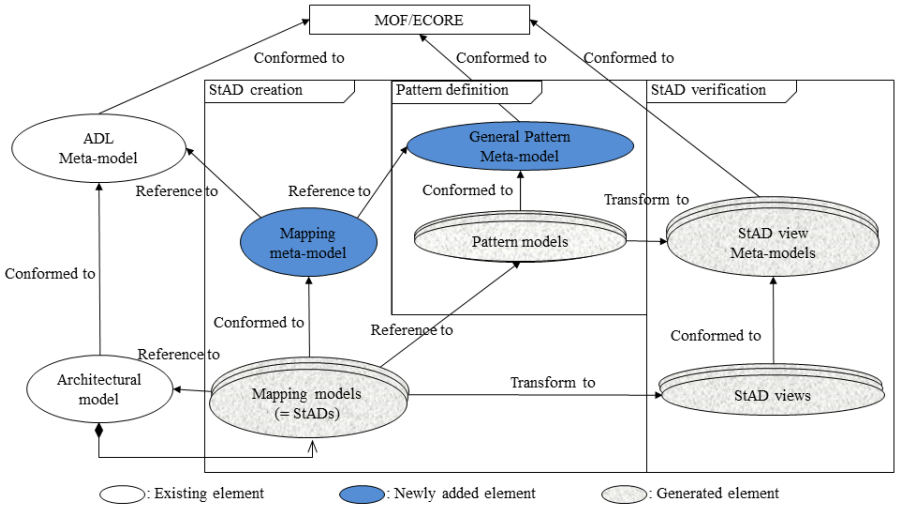

Fig. 5 MDA approach for StAD documentation

In the literature, architecture is considered as a set of views which are representations of system elements and relations associated with them [9]. Each view serves a specific purpose depending on the concerns of one or more stakeholders. Having taken this viewpoint into account, we propose to consider an *architectural model* as a multi-view representation where each view contains only elements related to a specific StAD. In this scenario, *StAD views* are filtered from the *architectural model* through a transformation mechanism in which *mapping models* are the source models and *StAD views* are the target models. In fact, we can consider the *mapping models* as the initial version of the *StAD views* where information about the integration of StAD is stored. Thus, the transformation is the step in which this information are concretized into *StAD views* elements.

4.3 StAD verification

To make sure that an *architectural model* is consistent with created StADs, not only do the existence of StAD-related elements in an *architectural model* need to be verified but also the constraints imposed on them need to be handled. To achieve the first goal, the presence of StADs in the *architectural model* is checked through the completeness of *mapping models*. Indeed, *mapping models* are intermediary bridges between the *architectural model* and the *pattern model* and thus, the incompleteness of *mapping models* shows the lack of StADs in the *architectural model*. To achieve the second goal, the constraints imposed by patterns on the *architectural model* are checked through the consistence of *StAD views*. To check the conformity of *StAD views*, we chose to first transform the *pattern models* into *StAD view meta-models* (Pattern verification part in figure 5) and then, make use of model checking techniques from MDA [27].

5 Pattern definition

The process of creating a pattern consists in instantiating a *pattern model* from its meta-model. We first introduce the *general pattern meta-model* from which pattern models are created. Then, we clarify the pattern definition process through a concrete example.

5.1 *General pattern meta-model*

The *general pattern meta-model* provides the language to define an architectural pattern. It contains all necessary architectural features from the target ADL and concepts related to pattern definition. Figure 6 illustrates two parts of the pattern language: pattern part and structural part. While the pattern part is general enough to be applied in any paradigm, the structural part represents concrete elements for each supported language family. The structural part of one paradigm can be replaced by the structural part of another one.

In this example we choose the structural part that corresponds to SOA description languages family. We provide more details about how to replace the structural part to switch to another language family in Section 8.1.

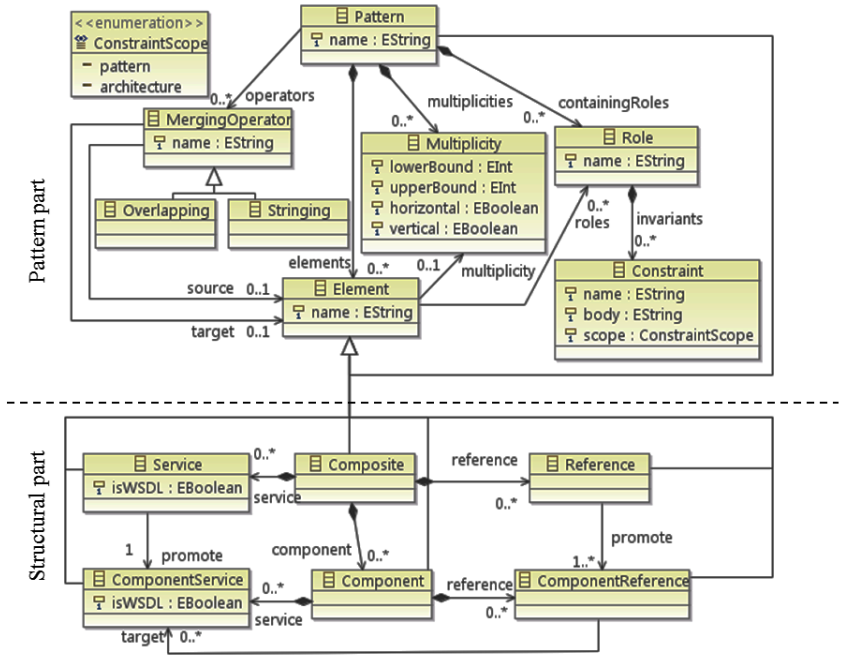

Fig. 6 SOA General Pattern Meta-model

Inspired by the SCA model[1] [4], we construct the structural part of our *General Pattern Meta-Model* for the SOA description language family as follows:

- *Composite* serves as the container to assemble and connect service-oriented building blocks together.
- *Components* are basic units of the architecture that represent business functions from which composite applications are built.
- A component is composed of *component services* and *component references*. The former provide functionalities supported by the component and the latter play the role of consuming services of other components. A component reference can be wired to a component service through its *target* attribute. The attribute *isWSDL* specifies whether the component service is a webservice or not.

[1] SCA is a model created by a group of industrial partners to support building applications and systems using SOA solution.

– Thinking of composites as black-box components, they have also *services* and *references*. To be consumed by the world outside of a composite, a component service of the containing component can be promoted as a service of the composite. Similarly, to be served by an outside service, a component reference should be promoted as a composite's reference.

The *pattern aspect* part of our meta-model aims at providing functionalities to characterize a meaningful architectural pattern. We reuse the composition-centered pattern description language that we proposed in another work [35]. To be more specific, the meta-model allows us to describe a pattern element at two levels: generic and concrete. Via the *multiplicity*, we can specify an element as generic or concrete. A concrete element (not associated with any multiplicity) provides guidance on a specific pattern-related feature. Being generic, an element (associated with a multiplicity) represents a set of concrete elements playing the same role in the architecture. A multiplicity indicates *how many times* a pattern-related element should be repeated and *how* it is repeated.

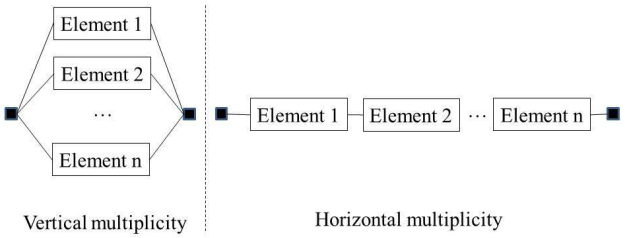

Fig. 7 Orientation organization of generic elements

Figure 7 shows two types of orientation organization for a multiplicity: vertical and horizontal. Being organized vertically, participating elements are parallel which means that they are all connected to the same elements, e.g. Clients are all connected to Server in the Client-Server pattern [6]. On the other hand, being organized horizontally, participating elements are inter-connected, e.g. Filters are connected to each other via Pipes in the Pipes and Filters pattern [6]. Each element in the meta-model can be associated with a *role*. A role specifies properties that a model element must have if it is to be part of a pattern solution model [11]. To characterize a role, we use architectural *constraints*. A constraint made to a role on an element helps to make sure that the element participating in a pattern has the aimed characteristics. Constraints are represented in our approach in form of OCL rules. The *scope* attribute of a constraint helps to determine the affected area of the constraint. If the scope equals *pattern*, the constraint is imposed only on elements contained in the pattern. Otherwise, the scope equals *architecture* means that the constraint involves not only elements in the pattern but maybe also other elements in the architecture. Let us take an example of a pattern containing two elements A and B with a constraint saying that *among all elements* connected to element

A, there is only element B assuring certain characteristics. This constraint has an *architecture* scope since it may involve external elements besides those defined in the pattern (A and B) to make sense. This attribute is important to determine pattern-related elements within the architectural model for the purpose of filtering pattern view (see Section 6.2). Patterns in our language can be composed using two operators: overlapping and stringing. A stringing operation means a connector is added to the pattern model to connect one component from one pattern to another component from the other pattern. An overlapping operation means that two involving elements should be merged to a completely new element. Figure 8 shows these two types of merging operation.

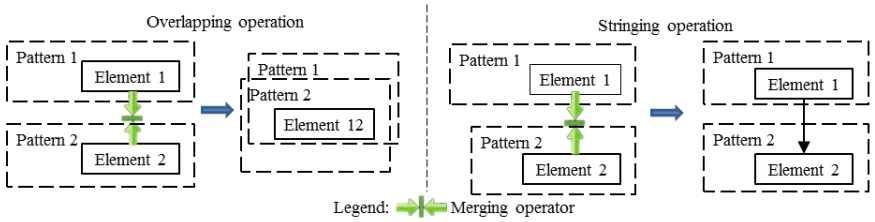

Fig. 8 Two types of merging operation [35]

The reader can find more information about how to compose patterns using these operators in [35].

5.2 Architectural Pattern Specification

For the purpose of illustration, we will examine the SOA Legacy Wrapper pattern [36] which is also mentioned in Section 3.

Based on the *general pattern meta-model*, we can instantiate the *pattern model* for the SOA Legacy Wrapper with the emphasis on the following elements (as illustrated in Figure 9): the component *LegacyComponent* specified with the role *LegacyComponent* representing the component with legacy implementations, the component *WrapperComponent* specified with the role *WrapperComponent* representing the wrapper services in the pattern. The component *LegacyComponent* is not assigned with any multiplicity since it represents a concrete legacy component. Otherwise, the component *WrapperComponent* is assigned with a multiplicity since it represents many possible wrapper components. Furthermore, its vertical multiplicity[2] indicates that there maybe many instances of WrapperComponent and they must be vertically connected. The role *LegacyComponent* is characterized by the *ShieldedByWrapper* constraint and the *OnlyConnectedToWrapper* constraint. The former stipulates

[2] upperbound = -1 indicates that there's no limited upper threshold for a multiplicity

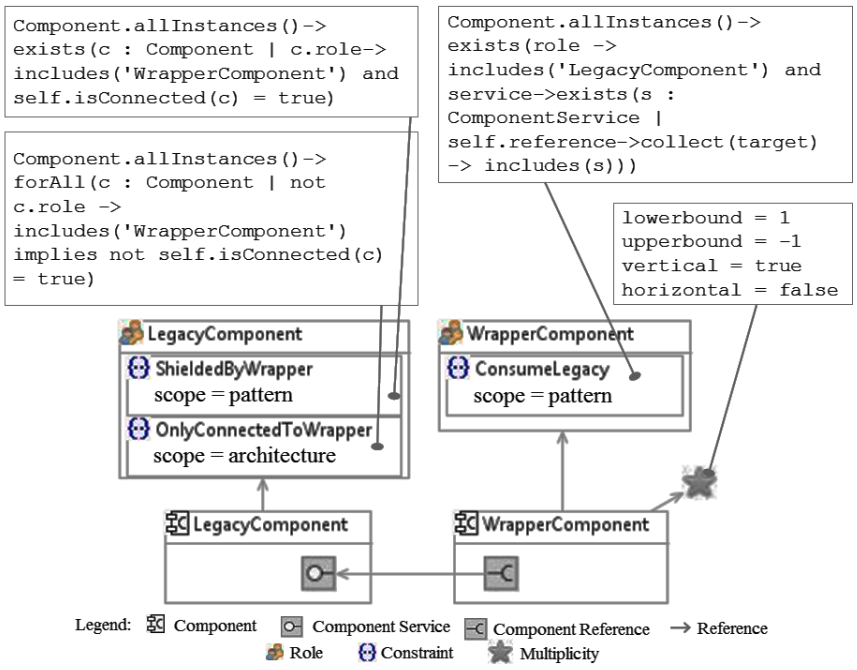

Fig. 9 SOA Legacy Wrapper pattern model

that there must exist a component that plays the role *WrapperComponent* and is connected to the legacy component, the latter stipulates that for all existing components, if one does not play the role *WrapperComponent* then it can not be connected to the legacy component. Note that the *ShieldedByWrapper* constraint has a *pattern* scope since it involves only elements playing either the role *WrapperComponent* or *LegacyComponent*. The *OnlyConnectedToWrapper* constraint has an *architecture* scope since it involves not only elements playing the two mentioned roles but maybe also other elements in the architecture. The last constraint *ConsumeLegacy* characterizes the role *WrapperComponent* and stipulates that there must exist at least one legacy component with all services wrapped by the wrapper component. This constraint also has a pattern scope. Even though the other participating elements in the *Legacy Wrapper* pattern model, such as the component service of *LegacyComponent* and the component reference of *WrapperComponent*, do not have specific roles, they still contribute to the model to make a meaningful pattern.

6 StAD Manipulation

The process of manipulating StADs consists of integrating pattern models to architectural models and checking the conformance of architectural models

with associated StADs. The former is made thanks to a mapping model and the latter is made thanks to a particular view on the architecture.

6.1 Associating a Pattern to an Architectural Model

The association of a pattern with an architectural model consists in manually creating a mapping model between the former and the latter. Concretely, it links elements in the architecture that directly relates to elements from the pattern model. For the sake of simplicity, we do not show the mapping meta-model here but basically, it consists of one meta-class per type of mapping. Each meta-class defines a mapping between the source, an architectural meta-class from the ADL meta-model and the target, an architectural meta-class from the *general pattern meta-model*. The reader can find this mapping meta-model in Appendix D

The architecture of FRC case study [36] is chosen to illustrate the documentation of pattern use. Figure 10 sketches the mapping model which associates the Legacy Wrapper pattern to a part of the FRC architecture. As we can see in this figure, two components *Data Controller* and *DWSA Data Service* in the FRC architecture are mapped respectively to two components playing the roles of *Legacy Component* and *Wrapper Component* in the *Legacy Wrapper* pattern model.

An architectural model can contain different mapping models, each of them represents a StAD made to the architecture. Thus, the architectural model can be considered as a set of elements in which each element can play different roles coming from the same StAD or different ones.

6.2 Filtering StAD views

The architectural views are useful for understanding the overall architecture of a complex system. In our case, each view represents one instantiation of a pattern in the architectural model. Thus, a view is produced by applying a transformation on the mapping model which captures a given StAD. The transformation serves as a filter to realize two purposes: first, extract from the architectural model elements related to StADs and second, eliminate language-specific features of the architectural model to create a language-independent pattern view model. This transformation can be compared to the one from PSM (platform specific model) to PIM (platform independent model) in the MDA approach. To realize this, we leverage the MDA transformation techniques. More precisely, we use Kermeta [26] transformation rules to transform mapping models into StAD view models.

Algorithm 1 illustrates the transformation of a mapping model into a view model. First, it detects whether the pattern model has an *architecture-scope* constraint. If not, the algorithm creates a StAD view model that contains only pattern-related architectural elements. These elements and their pattern

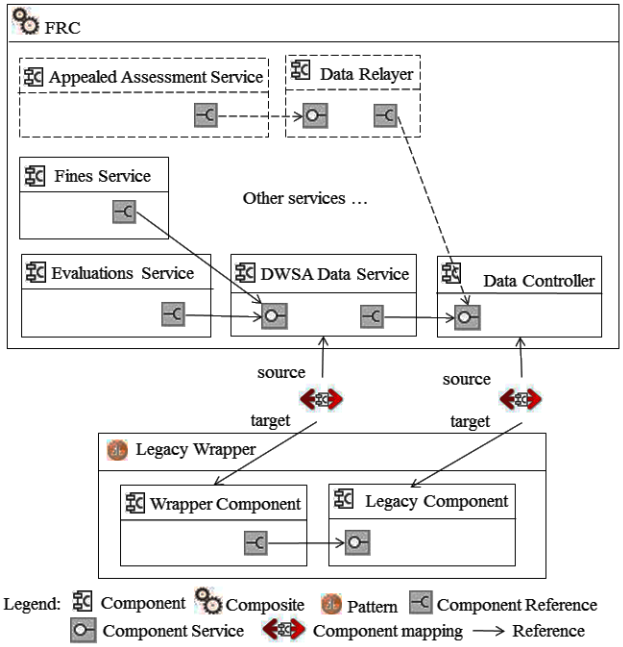

Fig. 10 Mapping model for the *Legacy Wrapper* pattern in FRC (Elements in dashed line are those added after the evolution of the FRC architecture)

roles are determined via the mapping model. Otherwise, if the pattern model has at least one *architecture-scope* constraint, then the algorithm creates a StAD view model that contains all elements in the architectural model. But, only the roles related to the concerned pattern are kept in the elements. Patterns with *architecture-scope* constraints are not usual, as shown through our experimentation on well-known SOA patterns (see section 8.1).

The bottom of Figure 11 represents the filtered StAD view model for the mapping model before (without elements in dashed line) and after (with elements in dashed line) the evolution of the architecture described in Figure 10. As we can recall from the *Legacy Wrapper* pattern defined in Figure 9, the scope of the constraint *OnlyConnectedToWrapper* is *architecture*. This leads to the creation of a StAD view that contains all elements in the FRC architecture with only their roles related to the *Legacy Wrapper* pattern. In the produced StAD view there are two elements holding a role in the *LegacyWrapper* pattern: *DWSA Data Service* and *Data Controller* playing respectively the two roles *WrapperComponent* and *LegacyComponent*. The choice of this illustrative pattern is made in order to cover the two kinds of constraint. In general cases architectural patterns hold mainly constraints with *pattern-scope*. To illustrate the StAD views generated in this case, we show on the top of Figure 11 the StAD view for the *Legacy Wrapper* pattern without taking into account its *architecture-scope* constraint.

Algorithm 1 The StAD view model filtering algorithm

Require: MappingModel MM
Ensure: StAdViewModel VM
1: PatternModel $PM \leftarrow MM.target$
2: ArchitecturalModel $AM \leftarrow MM.source$
3: Flag $f \leftarrow false$
4: **for all** Role $r \in PM.roles$ **do**
5: **for all** Constraint $c \in r.constraints$ **do**
6: **if** $c.scope = architecture$ **then**
7: $f \leftarrow true$
8: Break
9: **end if**
10: **end for**
11: **end for**
12: **if** $f = false$ **then** // No architecture-scope constraint is found // Only pattern-related elements are filtered
13: **for all** Mapping $m \in MM.mappings$ **do**
14: ArchitecturalElement $ae \leftarrow m.source$
15: PatternElement $pe \leftarrow m.target$
16: StAdViewElement $ade \leftarrow ae$
17: $ade.role \leftarrow pe.role$
18: $VM.add(ade)$
19: **end for**
20: **else** // At least one architecture-scope constraint is found // Filtering all elements
21: **for all** ArchitecturalElement $ae \in AM.elements$ **do**
22: StAdViewElement $ade \leftarrow ae$
23: **for all** Mapping $m \in MM.mappings$ **do**
24: **if** $m.source = ae$ **then**
25: $ade.role \leftarrow m.target.role$
26: **end if**
27: **end for**
28: $VM.add(ade)$
29: **end for**
30: **end if**

6.3 StAD Checking

The checking process consists of two steps: first, the completeness of the mapping model is verified and if the mapping model's integrity is assured, in the second step, the StAD view meta-model is used to check the consistency of the StAD view model. Whenever the mapping model is detected as incomplete (e.g. due to the removal of some StAD-related elements in the architecture) or constraints imposed by the StAD view meta-models on StAD views are not satisfied, warnings are notified to the architect about which StAD is violated and which elements in the architectural model are involved.

The conformity of an architectural model with its associated StADs is checked through the conformity of the corresponding *StAD view* with the concerned pattern model. For the purpose of checking, *StAD view meta-models* are generated from *pattern models*. The consistency of an *StAD view* is thus verified against its corresponding *StAD view meta-model* using the conformance operator from the MDA approach.

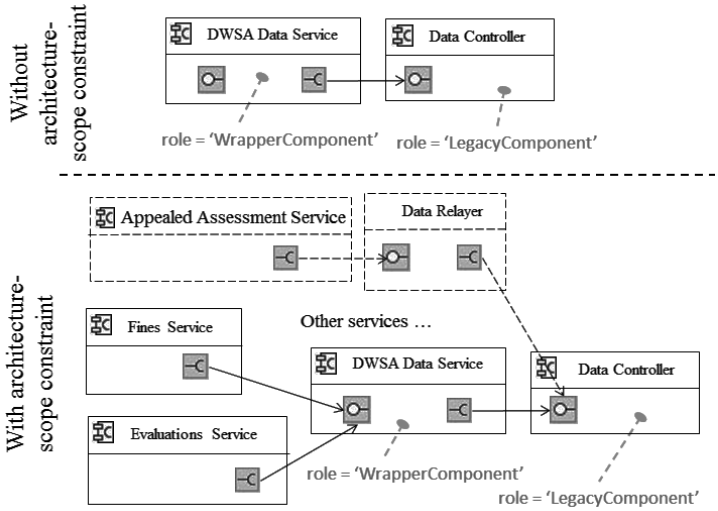

Fig. 11 StAD views for the Legacy Wrapper pattern produced from FRC architecture (Elements in dashed line are those added after the evolution of the FRC architecture)

For every defined *pattern model*, an *StAD view meta-model* is generated containing meta-classes from the *general pattern meta-model* embedded with pattern constraints. The Algorithm 2 represents the simplified version of this transformation.

Algorithm 2 The StAD view meta-model creation algorithm

Require: PatternModel M
Ensure: StAdViewMeta-model MM
1: $MM.add(createMetaClasses())$
2: **for all** PatternElement $pe \in M.elements$ **do**
3: Meta-class $mc \leftarrow MM.getCorrespondingMetaClass(pe)$ // Select the corresponding meta-class of the StAD view meta-model based on the pattern model element
4: **for all** Role $r \in pe.roles$ **do**
5: **for all** Constraint $c \in r.constraints$ **do**
6: $mc.add(c)$
7: **end for**
8: **if** $pe.getMultiplicity() \neq \emptyset$ **then**
9: Multiplicity $m \leftarrow pe.multiplicity$
10: $mc.add(createMultiplicityConstraint(m))$
11: **else** // Multiplicity constraint without parameter means that the meta-class should have exactly one instance
12: $mc.add(createMultiplicityConstraint())$
13: **end if**
14: **end for**
15: **end for**

For instance, the Legacy Wrapper pattern model described in the previous section will be transformed to an *StAD view meta-model* with the participa-

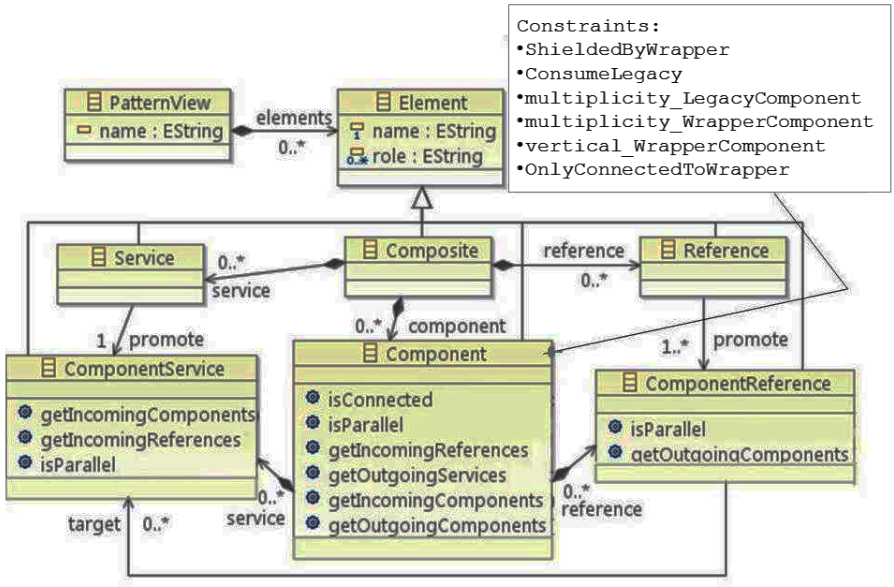

Fig. 12 StAD view meta-model for the Legacy Wrapper pattern.

tion of the following meta-classes: *Composite*, *Service*, *Reference*, *Component*, *ComponentService* and *ComponentReference* as shown in Figure 12. The StAD view meta-model is embedded with invariants imposed on the Component meta-class as follows:

```
(1) invariant ShieldedByWrapper:
  if role->includes('LegacyComponent') then
    Component.allInstances()->exists(c: Component |
    c.role->includes('WrapperComponent') and
    self.isConnected(c) = true)
  endif;

(2) invariant ConsumeLegacy:
   if role->includes('WrapperComponent') then
      Component.allInstances()->exists(role->
      includes('LegacyComponent') and service->
      exists(s: ComponentService | self.reference->
      collect(target)->includes(s)))
   endif;

(3) invariant vertical_WrapperComponent:
  if role->includes('WrapperComponent') then
     Component.allInstances()->forAll(role->
     includes('WrapperComponent') implies
     isParallel(self))
  endif;

(4) invariant multiplicity_LegacyComponent:
   let s: Integer = Component.allInstances()->
```

```
   select(role->includes('LegacyComponent'))->
   size() in s = 1;

(5) invariant multiplicity_WrapperComponent:
   let s: Integer = Component.allInstances()->
   select(role->includes('WrapperComponent'))->
   size() in s >= 1;

(6) invariant OnlyConnectedToWrapper:
  if role->includes('LegacyComponent') then
    Component.allInstances()-> forAll(c: Component |
    not c.role -> includes('WrapperComponent')
    implies not self.isConnected(c) = true)
  endif;
```

A part of invariants on meta-classes correspond to the constraints specified on the role elements in the pattern model. The other part reflects information about orientation and multiplicity. We can observe through the example that the constraints imposed on the roles *LegacyComponent* and *WrapperComponent* in the pattern model are transformed into the invariants *ShieldedByWrapper* (1), *OnlyConnectedToWrapper* (6) and *ConsumeLegacy* (2) on the *Component* meta-class. The multiplicity of the role *WrapperComponent* in the pattern model is concretized in two other invariants in the *Component* meta-class: *multiplicityWrapperComponent* (5) and *verticalWrapperComponent* (3). Since the role *LegacyComponent* in the pattern model is not associated with any multiplicity, the invariant *multiplicityLegacyComponent* (4) restricts the exact number of *LegacyComponent* in the pattern view to one.

The FRC architecture passed through an evolution in which two components *Appealed Assessment Service* and *Data Relayer* (sketched in dashed line in Figure 10 and Figure 11) are added. As we can observe, its mapping model is complete. However, the addition of the *Data Relayer* component violates the *OnlyConnectedToWrapper* constraint (6) since it is directly connected to a component playing the role *LegacyComponent*.

Similar to UML, this way of StAD specification allows us to introduce two levels of consistency: meta-model and well-formedness rules. More specifically, well-formedness rules are expressed in OCL to assert the syntactic correctness of StAD view models. According to the classification of model consistency methods presented in [22], this approach is a syntactic-horizontal consistency one.

7 Implementation

To verify the feasibility of our approach, we developed a tool called *ADManager*, which in its actual version, supports the documentation of StADs in three different languages: SCA [4], Acme [13] and PiADL [29].

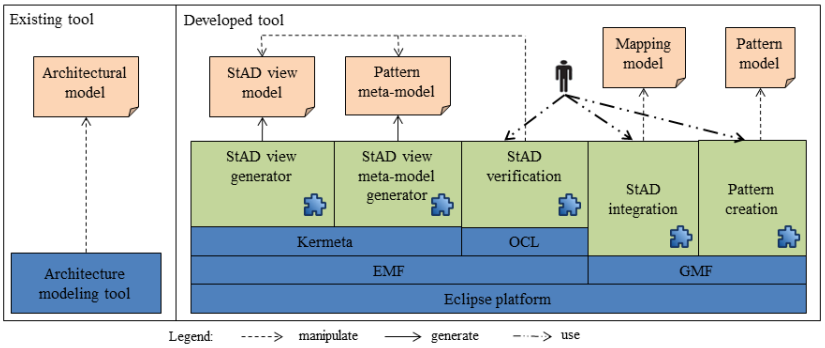

Fig. 13 The architecture of ADManager

7.1 *ADManager* tool

With *ADManager* we aim to make concrete the aforementioned concepts. The tool provides the following functionalities:

1. Create architectural patterns
2. Integrate StADs to architectural models
3. Verify the consistency of architectural models according to the held StADs.

ADManager is developed based on EMF (Eclipse Modelling Framework) [32]. We choose EMF to realize our tool since we leverage MDA, where models are basic building units, to develop our approach. As shown in Figure 13, the tool consists of five Eclipse plug-ins built on existing Eclipse technologies. They are:

- *Pattern creation plug-in* uses EMF and GMF (Graphical Modeling Framework)[3] modeling support in order to allow architects to define *Pattern models* graphically.
- *StAD integration plug-in* is an editor supporting the creation of *Mapping models* between pattern elements and architectural model elements.
- *StAD verification plug-in* uses OCL tool to support writing rules in pattern models, during pattern creation, as well as conformance verification between StAD view models and StAD view meta-models during StAD checking.
- *StAD view meta-model generator plug-in* uses Kermeta to implement rules generating StAD view meta-models from pattern models.
- *StAD view generator plug-in* uses Kermeta to implement rules generating StAD views from mapping models.

Note that in Figure 6 the *General Pattern Meta-Model* for SOA is separated into two parts: one specific to the SOA description language family and the

[3] `http://www.eclipse.org/modeling/gmp/`

other for the notion of pattern. The separation of these two aspects gives our tool the flexibility to support many different ADL families just by switching to the appropriate structural part of the pattern meta-model. Indeed, besides SCA, we have been able to support two different ADLs, namely Acme and PiADL, by keeping the pattern part and modifying the structure part in the pattern meta-model[4].

As *ADManager* is also a consistency management tool, it supports the following functionalities among those described in [10]:

- Automatic inconsistency detection - The tool automatically extracts pattern view models from mapping models and checks their consistency.
- Visual inconsistency presentation - The tool provides a visual inconsistency feedback to the user via description dialogues.
- Inconsistency diagnosis - The tool diagnoses a model and presents a user with a summarized report about which pattern is violated and if so, at which constraint.

The reader may obtain a complete guiding tutorial video and more information about the *ADManager* tool at: `http://www-archware.irisa.fr/software/admanager/`.

8 Evaluation

The main contribution which lies behind our work is the documentation of pattern-centric StADs which maintains both the existence of StADs and their structural consistency. For this purpose we defined a general pattern definition language that can be switched from one paradigm to another. Thus, to evaluate our approach, we first show the ability to adapt our pattern definition language in two different paradigms. Then we show the effectiveness of StADs'documentation and their completeness in terms of existence and structural consistency.

8.1 Application of pattern definition language

To evaluate the support of multi-paradigms, we collected patterns from two different paradigms, namely SOA and Component-Based Architecture (CBA), and see how our pattern definition language can support them. There are two criteria upon which patterns are chosen to be formalized. The first one is that the pattern's vocabulary cannot extend beyond the concepts supported by the corresponding ADL. The second one concerns the scope of the pattern. We limit selected patterns to the structural aspect, other patterns are considered to be out of scope.

[4] The reader can find these meta-models at the same website of ADManager tool

8.1.1 SOA patterns

We have examined the SOA patterns from [36] that are summarized in Table 1. In that table we reused the categorization of patterns given in [36]. Among the 80 identified patterns there are up to 50 patterns focusing on the aspect of service management. Examples of service management patterns are those concern how to physically centralize or decentralize services, how to determine the boundary of service logic, etc. These patterns in fact do not directly concern the structural aspect of the architecture. Therefore, they cannot be formalized using concepts from the ADL.

Table 1 Categories of SOA Patterns from [36]

Pattern category	Patterns	Architectural patterns	Formalized patterns	Patterns with architecture scope constraints
Service inventory design patterns	24	10	5	0
Service design patterns	33	14	14	2
Service composition design patterns	23	6	6	2
Total	80	30	25	4

As we can observe in the Table 1, among the remaining 30 architectural patterns there are ones based on architectural concepts that are not supported yet by Service-Oriented ADLs such as service inventory, service layer, etc. This explains why only 25 patterns are formalized using our approach ("Formalized patterns" column). Most of the formalizable patterns fall into the Service design pattern category. Indeed, patterns in this category are good practices in service organization, encapsulation, implementation, governance and therefore, suitable to be architecturally formalized. The column "Patterns with architecture scope constraints" gives the number of patterns holding at least one constraint with architecture-scope. Only 4 patterns, among the 25 formalizable ones, fall in this case.

8.1.2 CBA patterns

To support the design of CBA patterns, starting from the SOA pattern meta-model (see Section 6), we switched the structural part to another one that conforms to CBA patterns. Figure 14 shows the CBA pattern meta-model. More specifically, this structural part consists of the set of architectural elements: component, connector, port and role. This set of elements is in fact the necessary design vocabulary for architectural pattern as pointed out in [25,9]. Switching from the SOA pattern meta-model to the CBA pattern meta-model

is in fact the matter of disconnecting the structural part of the former and connecting the structural part of the latter. More specifically, connecting the CBA meta-model to the pattern part consists in i) Adding the inheritance relationship between two meta-classes SimpleElement and CompositeElement (CBA part) and the Element meta-class (Pattern part) and ii) Adding a composition relationship between the CompositeElement meta-class (CBA part) and the Element meta-class (Pattern part). Thus, the switching is realized without any modification in the pattern part of the general pattern meta-model.

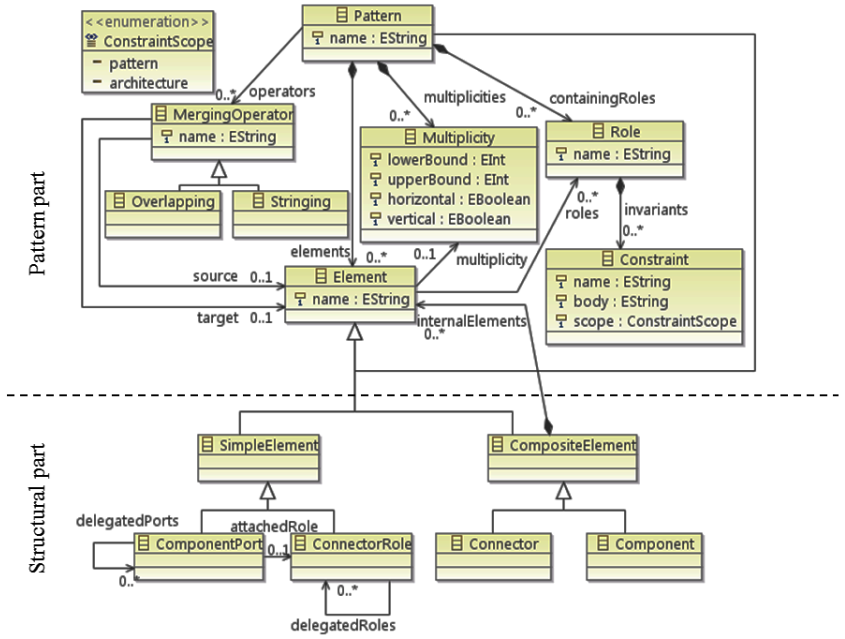

Fig. 14 General CBA pattern meta-model

Using this adapted pattern language, we have tried to model the CBA pattern catalogue described in [2]. Table 2 shows the examined patterns assigned to different viewpoints.

As we can observe, we have been able to model 14 patterns (Column Structural patterns) out of the total 24 patterns (Column Pattern). Our pattern definition language focuses on the structural aspect of architectural patterns. Thus, the patterns concerning the behavioural aspect are not formalized in our study. Representatives of behavioural patterns can be those dealing with invocation mechanism, runtime events, etc. None of the 14 formalized patterns contain a constraint with architecture scope.

The reader can find a complete list of formalized patterns in two paradigms SOA and CBA in Appendix A and Appendix B.

Table 2 Categories of architectural patterns from [2]

Architectural view	Pattern	Structural pattern	Pattern with architecture scope constraints
Layered view	2	2	0
Data flow view	2	1	0
Data-centered view	3	3	0
Adaptation view	3	2	0
Language extension view	3	0	0
User interaction view	3	3	0
Component interaction view	5	2	0
Distribution view	3	1	0
Total	24	14	0

8.2 StAD documentation

The following subsections discuss the effectiveness and the completeness of our support for StAD documentation. We first introduce the materials used in our study and then go into details of evaluation results.

8.2.1 Evaluation materials

We evaluated our approach with 8 architectural models. These models vary in terms of size and domain. They are gathered from different sources in the literature. These models as well as the evaluation results can be found in a technical report entitled IRISAArchWare-2013-TR-02[5] at ADManager's website. We choose Acme as the ADL to depict these models in this evaluation but as we stated in section 7.1, it is feasible to change to another ADL since the pattern meta-model is language independent. In Acme, an architecture is described using elements such as *component, connector, port, role, representation, attachment, etc.* among which components and connectors play the most crucial roles. Thus, the size of model is expressed according to three types of measurements: first, by the number of all elements, second, the number of components and third, the number of connectors. Figure 15 shows the sizes of the models in terms of model elements, components and connectors. They differ from small models (49 elements, 7 components and 6 connectors) to big models (287 elements, 34 components and 36 connectors). The models cover different domains, from source code management systems, digital publishing systems to software product line middlewares, etc. The reader can find the list

[5] https://www-archware.irisa.fr/files/2014/05/IRISAArchWare-2013-TR-021.pdf

of architectural models together with applied patterns and their frequency of usage in Appendix C.

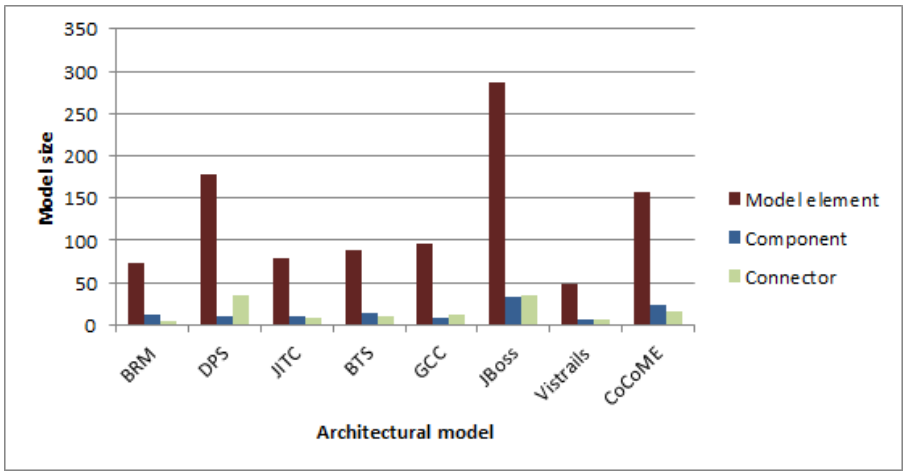

Fig. 15 Size of 8 Acme architectural models in terms of model elements, components and connectors

One important criterion in choosing these models is that they must focus on the application of architectural patterns in their design. All of the chosen models are indeed designed using architectural patterns from different paradigms such as Enterprise Integration, SOA, etc. On average, there are two patterns applied per model.

8.2.2 Modeling effort and StAD violation detection

We present in this section a quantitative evaluation on the modeling effort of using our approach and how this effort is paid off through the detection of StAD violation. As we can recall from section 6, the advantage of using mappings is twofold: i) They serve as the bridge between the architectural model and the pattern and thanks to this, the pattern language is independent from any ADL; ii) They are a means to stock the decision of applying a pattern. The question raised is how much effort do we afford to create these mappings for this aim. We count the number of all mappings for each pattern applied in an architectural model and compare it to the number of model elements to determine whether the mappings would not overwhelm the architects. Figure 16 shows the size of mappings comparing to size of architectural models in terms of components and connectors. We found that the number of mappings is in average 9.12 (between 3 and 22) and moreover, the average #mappings/#elements is 26.11%, which is a reasonable number. In fact, seeing that mappings take part in the documentation of StADs, the question of

adding mappings or not can be considered as the trade-off of documenting a StAD. This trade-off has been also discussed in [40, 44].

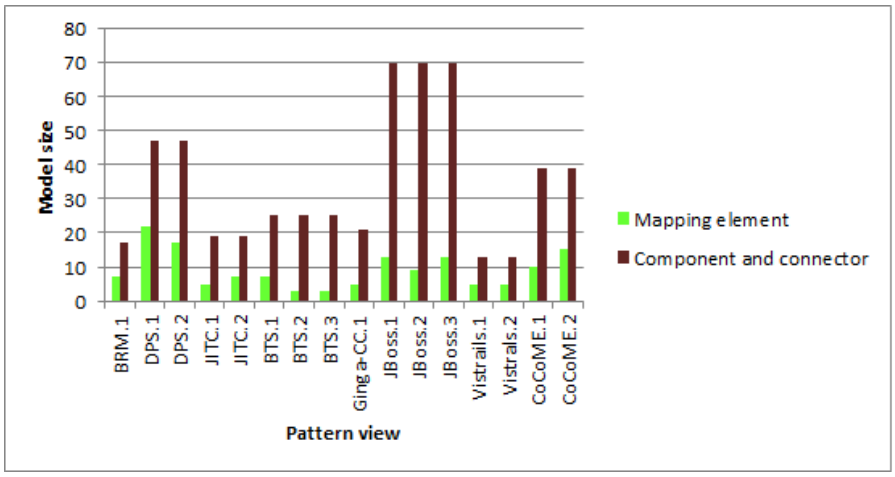

Fig. 16 Size of mappings comparing to size of architectural models in terms of components and connectors

The first benefit gained from our approach is the obtainment of simplified pattern views. Pattern views serve as a filter of pattern-concerned elements from the architectural model. We evaluated whether pattern views help reduce a great number of non-related elements and thus, improve the understanding of the created StAD. Figure 17 shows the comparison between the size of pattern view and the size of the architectural model where it is drawn from. We can observe that most of the pattern views filter out less than 30% of model elements. Two exceptions are the first pattern view of BRM (49 pattern view elements over 73 model elements, equivalent to 67%) and the second view of DPS (91 pattern view elements over 179 model elements, equivalent to 51%). The reasons for this is that the BRM architecture is constructed using *Layers pattern* as the basic principle. Thus, most of the elements participate in the *Layers pattern view*. Similarly, most of the components in the DPS architecture play the role of *Data accessors* in the *Repository pattern*. If we consider all pattern views, pattern view elements are about 25% of the total number of elements in average.

The second benefit of our approach is a complete mechanism of StAD violation detections. Our approach emphasizes the combination of mapping models and pattern models in documenting a StAD. Indeed, mapping models and pattern models together maintain the existence of the StAD and its structural consistency. The absence of one of these two artefacts will lead to an incomplete StAD and thus, undetected violations. To confirm this remark, we

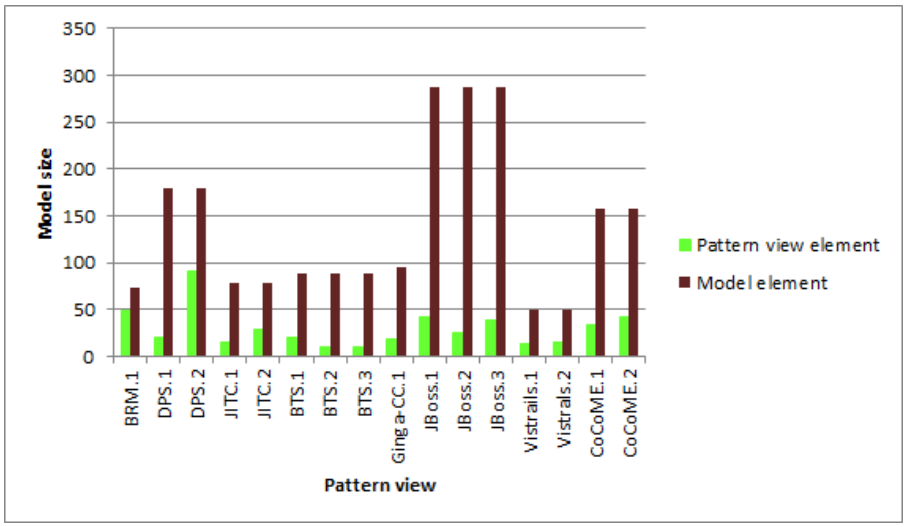

Fig. 17 Size of pattern view comparing to size of architectural models

make the architectural models evolved and see if we can detect the violations in two cases: i)without mapping models and ii)without pattern models.

An evolution of an architectural model can fall into two cases: deletion and addition of elements (modification is the combination of these two operations). We do not have architects' participation to set up real-life scenarios which involve only meaningful evolutions. Instead, we randomly seed deletion and addition of architectural elements.

Table 3 Deletion of architectural elements

Model	Nb of combinations of meaningful deletions	Nb of pattern-related deletions	Nb of detected violations by mapping	Nb of detected violations by pattern	Nb of detected violations by our approach
BRM	157	22	22	16	22
DPS	68719478782	776	776	518	776
JITC	258	32	32	22	32
BTS	3076	30	30	21	30
GCC	4606	10	10	7	10
JBoss	271336	426	426	364	426
Vistrails	190	20	20	14	20
CoCoME	488	460	460	327	460

Table 3 shows the result in the case we delete architectural elements. Theoretically, if N = Nb of components + Nb of connectors, then the total number of combinations of possible deletions is

$$\sum_{k=1}^{N} \binom{N}{k} = 2^N - 1 \tag{1}$$

This is in fact the sum of every combination of components and connectors. If we take the case of the biggest model (34 components and 36 connectors), this sum is up to around 10^{21} possible deletions. Among these deletions, we exclude those when the components are deleted but their associated connectors remain. It of course makes no sense of a deletion to leave a dangled connector in the model. Taking this condition into consideration, the number of possible deletions is reduced significantly and is reported in the column 2 (Nb of combinations of meaningful deletions) of Table 3. Particularly, the DPS model has up to 68719478782 possible combinations of deletion because its architecture resembles to that of a strongly connected graph where each component has connectors to many other components. Among these deletions, those concerning the application of pattern continue to be filtered out (column Nb of pattern-related deletions). All of these deletions violate the StAD about using pattern (detected by *mapping models or pattern models*). It is clear that 100% violated pattern-related deletions can be detected by mapping models (column Nb of detected violations by mapping) because the latter binds to every element concerning the former. What is noticeable is that there are a certain number of pattern-related deletions that can not be detected by pattern models. As we can observe from the column 5 (Nb of detected violations by pattern), the number of detected violations by pattern models is lower than the total pattern-related violations. In average, 72% of pattern-related violated deletions can be detected by pattern models and the rest 28% can not be detected. The undetected cases of violation by pattern are shown in our aforementioned technical report. One example of these could be the case when the first or the last Filters in the Pipes And Filters are deleted. In this case, the remaining Filters and Pipes would make a perfect Pipes And Filters pattern without knowing that some Filters and Pipes have been deleted. In other words, the decision about using the Pipes and Filters pattern has been affected while the pattern model itself cannot recognize it. Another example is the case when the entire pattern (whatever pattern) is deleted. The pattern model is useless since its instances disappeared, leaving the task of keeping track of the decision of using pattern to mapping model. Our approach (last column) detects all violations that were detected by mapping models.

Table 4 shows the result in the case we add elements to the architectural models. Since we focus our evaluation on architectural patterns, where the most significant modifications happen at a coarse granularity level (component and connector), we consider only additions of component and connector. Among them, we continue to limit the additions to those of connectors between existing components (the combinations of addition are not taken into consid-

Table 4 Addition of architectural elements

Model	Nb of total additions	Nb of pattern-related additions	Nb of detected violations by mapping	Nb of detected violations by pattern	Nb of detected violations by our approach
BRM	30	6	0	3	3
DPS	55	39	0	0	0
JITC	30	9	0	3	3
BTS	57	8	0	5	5
GCC	36	3	0	3	3
JBoss	145	52	0	30	30
Vistrails	21	6	0	3	3
CoCoME	44	34	0	24	24

eration). The reason for this limitation is that unlike the case of deletions where the number of simulated deletions are finite (seeing that the number of existing elements is fixed), the number of additions is infinite (seeing that we can arbitrarily add elements to the architecture). Besides, the change of an elements definition (name, type, etc.) is considered as the deletion of the element and the addition of a new element (old element with its new definitions). Thus, we took the most basic cases of deletion [6]. The number of the possible additions is reported in column 2 (Nb of total additions). Only a part of these additions relates to patterns. These pattern-related additions are shown in column 3 (Nb of pattern-related additions). Column 4 (Nb of detected violations by mapping) shows that none of the violated additions can be detected by mapping. This is true because the integrity of mapping models is always maintained despite of any addition and thus no violation can be detected. Otherwise, the additions that affect the structural consistency of the applied pattern will be detected by pattern model. As shown in column 5 (Nb of detected violations by pattern), pattern models can detect a part of violated pattern-related additions. In average, 53% of pattern-related additions are violated and detected by pattern models. The violated cases of addition are shown in the aforementioned technical report. One example of them could be when we add a Pipe between two distant Filters to create a cycle which is not permitted in Pipes and Filters pattern. Our approach (last column) detects all violations that were detected by pattern models.

The two above evaluations show that depending on the applied patterns, there are violations that can only be detected by the mappings but not by the pattern model and vice-versa. This is also the point that makes our approach stand out from the existing works which focus either only on the existence of ADs element or the structural consistency of ADs element. We combine both mapping, which assures the existence of ADs element, and pattern model,

[6] This is also discussed as a threat to validity in Section 8.2.3

which assure the structural consistency of ADs element, to verify StAD and thus, all violations are detected.

For the purpose of providing enough information to replicate the evaluation, in the following we go to some details with examples about the evaluating process as well as the measurements. Considering the case of the first evaluation (e.g. the deletion seeding evaluation), we first proceed with the measurement of the number of combination of meaningful deletions of a model. A meaningful deletion is one that involves either a connector or a component and does not leave a dangled element. Since a connector is always connected to two components, the deletion of a single component will always leave at least one connector dangled. Thus, a meaningful deletion of a component must affect all of its connectors. Figure 18 shows an example of a simple architectural model (on top of the figure) with three components: *Comp 1*, *Comp 2* and *Comp 3* and two connectors: *Con A*, *Con B*. On the bottom right of the figure is a meaningful deletion, in which *Comp 2* and its associated connector *Con A* are removed respectively. At the bottom, in the middle is the case when *Comp 1* and its surrounding connectors *Con A* and *Con B* are deleted. Finally, the bottom left of the figure is the case when *Comp 3* and *Con B* are deleted. From these 3 cases of component deletions we can create 7 combinations of component deletions (e.g. three 1-combinations, three 2-combinations and one 3-combination).

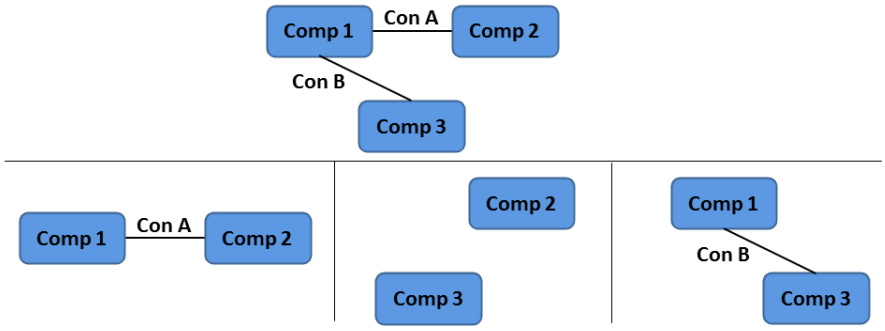

Fig. 18 Example of meaningful deletions

Thus totally we have 5 meaningful deletions (2 deletions of connectors and 3 deletions of components). From these 5 deletions we can create 10 different combinations of deletions (e.g. 3 combinations of connector deletions and 7 combinations of component deletions). If we take all possible deletions into consideration, there are up to 31 combinations of deletions (e.g. from 1-combinations to 6-combinations). This is to see how the filter reduces the number of possible deletion combinations.

Next, two scenarios are set up: the first one only involves mappings in violation detection and the second one only involves pattern model. First, let

us assume that our example is a *Client-Server* pattern where *Comp 1* is the Server and *Comp 2, Comp 3* are the Clients. In the first scenario, every element in the model is mapped to the AD. Thus, every deletion of mapped element is considered a violation. For instance, in our example, the deletion of *Comp 2* and *Con A* will be counted as a violation since both of them have been mapped. Thus, three of possible deletion combinations will be detected as violation. In the second scenario, pattern-related elements are directly assigned with roles and the conformance is checked against a pattern meta-model. For instance, the deletion of *Comp 1* will trigger a violation since the pattern constraint is not assured. However the deletion of *Comp 2* and *Con A* will not be detected as a violation since the rest of the model (e.g. *Comp 1*, *Con B* and *Comp 3*) will make a perfect Client-Server pattern. Among the three combinations of deletion, only one is detected (the case when *Comp 2, Comp 3, Con A, Con B* are all deleted and leave no Clients) with the pattern meta-model. Our approach combines these two ways of detecting deletion violation and thus, can detect three cases of violation.

Algorithm 3 The evaluation algorithm

Require: Model M
1: Set of meaningful connector deletions $SetConDels \leftarrow \{\}$ // An empty set of deletions of connectors
2: Set of meaningful component deletions $SetCompDels \leftarrow \{\}$ // An empty set of deletions of components
3: **for all** Element $e \in M.elements$ **do**
4: **if** e is $Connector$ **then**
5: $SetConDels \leftarrow e$ // Add the Connector in question to the set of meaningful connector deletions
6: **end if**
7: **if** e is $Component$ **then**
8: Set of related elements $SetEles \leftarrow getRelatedElements(e)$ // Derive a set of the Component and its surrounding Connectors
9: $SetCompDels \leftarrow SetEles$ // Add the the set of Component-related elements in question to the set of meaningful deletions
10: **end if**
11: **end for**
12: Set of meaningful deletion combinations $SetDelCombs \leftarrow getCombinations(SetConDels) \cup getCombinations(SetCompDels)$ // Derive the set of combinations of meaningful deletions of connectors and components
13: **for all** Set $s \in SetDelCombs$ **do** // From each combination, perform the deletions and verify the model's conformity
14: Perform the deletions based on s
15: Verify the conformity of the model using mappings
16: Verify the conformity of the model using pattern
17: Verify the conformity of the model using mappings and pattern
18: **end for**

Algorithm 3 shows the skeleton of the evaluation process. From an architectural model, we try first to determine the set of meaningful connector deletions. Next, we create a set of meaningful component deletions (for each given component taking into consideration surrounding connectors). Then thanks to a

combination generation algorithm, we obtain a set of combinations of meaningful deletions (including connector deletions and component deletions). For each combination of meaningful deletions we apply to the model and verify its conformance using mappings, pattern and both, respectively. Note that the manipulation of model and its elements, and the measurement are done thanks to EMF and its accompanied technologies. We do not show an example of addition seeding and violation detection here but the set-up and applying scenarios remain the same.

8.2.3 Threats to validity

This section discusses the study's various threats to validity.

Internal validity: Internal validity is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variable. In the case of seeding deletion operations, our independent variable is the effect of detecting violations with/without our approach. Similarly, the same measurement is performed in the case of seeding addition operations. We excluded meaningless deletions to reduce the number of treated deletions. However, in case of meaningful deletions and meaningless deletions mixed together, the approach can take more effect. We also chose to separate these two independent variables to highlight the drawbacks of using mapping models or pattern models independently. However, in case of seeding deletions and additions one after another, the effect of one independent variable can affect the other one and vice versa.

External validity: In the study, we simulate the architecture's evolution by seeding modifications (deletions and additions). Except for deletion operations when we can determine all possibilities of deleting elements in a model, the addition operations are unpredictable. In the study we treated only additions of connectors between components because the number of these additions is finite. Moreover, deletions and additions are not the only cases of architecture evolution, there are also other types of evolution at the finer granularity level, e.g. elements can be renamed, a connector can change type, and so on. Thus, this evaluation cannot generalize the effect of our approach in the evolution of architecture in general. However, the more cases of evolution, the more likely StAD violations are detected by mapping models and pattern models, and thus the more likely the approach has effect.

9 Related work

Our work directly concerns three aspects: definition of ADs, AD conformance checking and pattern-oriented architecture. Thus, in the following we will discuss work related to these three aspects.

9.1 AD Documentation

In the literature there are many proposed models and tools supporting AD documentation. Among these works, we can mention some representative models such as the architectural decision template [40], the ontology of design decisions [20] or recently the MAD 2.0 model [42], and tools such as Archium [17], ADDSS [7], AREL [33]. We can observe that most of these works concentrate on capturing and characterizing ADs but little focus is paid on the automated checking of design decision compliance in architectural models. In our work, we do not attempt to just define another AD model but propose a way to define StADs which allows to automatically detect their violation in the architectural model.

In [46], Zimmermann et al. point out the importance of reusable ADs in decision identification, decision making and decision enforcement and propose a model to document reusable ADs. Furthermore, in [47], they propose to weave pattern information into reusable architectural decision models to benefit their mutual interests. Besides that, in [14], Harrison et al. compare pattern and AD and think that the former can be leveraged to document the latter. These ideas focus on the fact that pattern use is an important information that completes the AD. Having taken them into consideration, we go further with our approach, which add the pattern formalization into the representation of StADs.

9.2 AD Conformance Checking

Being one of the first work dealing with StAD conformance checking, Tibermacine et al. [37,38] propose a family of architectural constraint languages to describe the structural part of StAD. Architectural constraints are used as a means to formalize StADs. With our approach we raise the level of abstraction by using architectural patterns to document StADs. StADs are no longer architectural constraints imposed on the architectural model but self-contained semantic pattern models. Thus, with a library of architectural patterns (defined with our approach) the architect does not have to directly manipulate constraints to define StADs.

The topic of *StADs about the use of patterns* has become an interesting subject in researches about ADs. Patterns are considered to be important sources to offer an effective way of capturing StADs [15]. The notion of pattern-centric StADs have also been used in [43,45] to emphasize the documentation of decisions when applying patterns in the architecture. The idea consists in leveraging the information contained in the applied pattern to inexpensively document the StAD about the application of pattern. In these approaches, similar to the description of patterns, the documentation of StADs about pattern use also exists in textual form. Another way of documenting StADs about pattern use is to construct StADs in form of linking elements between the StAD model and pattern-related elements in the architectural model [19,21,18]. Contrary

to the textual form, this form of representing StADs can be used to automatically check the conformance of StADs. Indeed, the idea of linking StADs and architectural artifacts to maintain the consistency during their co-evolution has been discussed in [8,19,21,23]. While the explicit link in [8] only serves for the traceability from decisions to design artifacts, the linking model defined in [19,21,23] also aims at checking the conformance of the architectural model. More specifically, in [19], the authors propose to reinforce the outcome of StAD by using model differences. The outcome of StAD is considered as a set of model changes. Model changes serve as bindings between affected model elements and model differences. The architectural model is consistent with made StADs iff affected model elements prevail. Similar to [19], in [23], the authors influence the outcome of StAD by using actions. A StAD is provided with a set of actions which in turn are concretized into design model elements. Particularly, checking rules are automatically derived from the actions via a transformation mechanism. The common point of these two works is that reusable StAD is represented by a set of changes in the architectural model. The architectural model is said to be consistent with StAD as long as these changes prevail. This can be compared to our work as if the checking is realized based solely on the completeness of the binding model. However, in our approach, the conformance of the StAD against its architectural model is checked through not only the mapping model but also the rules imposed by the StAD model itself. Indeed, by using pattern to describe the StAD, we make sure that it is reflected semantically in the architectural model. In [21], the authors propose to impose OCL constraints at model level to insure StAD. Many different decision types are introduced, each one is represented by an OCL rule. We think that it is more general and complete to represent StAD by pattern model. Actually, a pattern itself can be considered as a decision type which encompasses structures and rules. Please also note that the general idea of this proposal is presented in our previous work [34]. In this paper, we elaborate the presented approach and conduct an evaluation on it.

9.3 Pattern-oriented Architecture

Our idea of using a role-based approach to model patterns is inspired by the work of France et al. [11]. The authors propose to incorporate role information at the UML meta-model to facilitate the use of design patterns. Our approach is not just limited for object-oriented patterns but other language families such as architectural patterns, service-oriented patterns,... can be applied as well. Existing pattern-oriented ADLs such as Wright [1] or Acme [13] propose to build the architecture based on style specifications. Architectural elements are instantiated from architectural types and the consistency of the system is checked through constraints written at the architectural style level. However, in these languages the trace of the use of architectural styles is not preserved. In other words, the consistency of the style applied in the system is checked only if its instances exist. Thus, in case of removing elements involved in a style

(representing an StAD) there are some violations that cannot be alerted to the architect during an evolution of the architecture (as shown in our study). In our approach, we make explicit the link between the pattern and its instances and, as we have shown throughout the paper, it contributes an important part in StAD's documentation.

10 Conclusion

To address the issue of additional costs caused by the non-explicitness of the ADs, we proposed a solution where the ADs are not only explicit, but also first-class elements in the architecture definition. We specifically focused on ADs related to pattern application. Our approach leverages the combination of mapping models and formalized architectural pattern models. This combination brings two major advantages: i) it increases the level of AD reuse during the design stage, ii) it allows automate checking of the existence of ADs that must be maintained after the architecture's evolution. More importantly, we show that this is the more complete way comparing to related work to detect StAD violation. We have implemented our approach through a pattern definition language, a process and a tool to automate the checking of the architecture's consistency, with respect to the concerned ADs, during its evolution.

We used the MDA approach for two main purposes: i) During the definition of the pattern description language, this has allowed us to make a clear separation between the concepts specific to patterns and those specific to the architecture. This separation aims at making our pattern description language easily adaptable to various ADLs of different paradigms as shown through our evaluation. ii) To check the conformance of the architecture with respect to certain patterns, the model refinement mechanism has facilitated the extraction of views targeting the concerned patterns.

With our approach a company can build its own library of patterns representing some of its accumulated best practices. Thus, it becomes possible to automatically check that the produced architectures conform to the defined best practices. However, if the company uses different ADLs with different paradigms, or decides to move to an other paradigm, it will take an effort to redefine all existing patterns to fit the new paradigm. But, there are often common patterns in different paradigms (e.g. pipe and filter). This is a limitation in our approach of pattern description. A solution to this limitation would be to describe, in a generic manner, patterns that do not rely on a particular paradigm. Then, provide a means for their projection in each paradigm to avoid the redefinition of pattern. This is one of our future work.

Even though we validated that our approach is not limited to a given paradigm and that it detects more violations than the existing approaches, it still needs further validation about its potential extra cost and the acceptance by architects. To achieve this kind of validation, we need a controlled experiment. This is one another future work.

References

1. R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, 1997.
2. P. Avgeriou and U. Zdun. Architectural patterns revisited a pattern language. In *In 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee*, pages 1–39, 2005.
3. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, 2003.
4. M. Beisiegel, D. Booz, S. TIBCO, M. BEA, C. Sharp, and Y. SAP. SCA service component architecture. *Assembly Model Specification*, 2007.
5. J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
6. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.
7. R. Capilla, F. Nava, S. Pérez, and J. C. Dueñas. A web-based tool for managing architectural design decisions. *SIGSOFT Softw. Eng. Notes*, 2006.
8. R. Capilla, O. Zimmermann, U. Zdun, P. Avgeriou, and J. M. Küster. An enhanced architectural knowledge metamodel linking architectural design decisions to other artifacts in the software engineering lifecycle. In *Proceedings of the 5th European Conference on Software Architecture*, pages 303–318. Springer-Verlag, 2011.
9. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond (2nd Edition)*. Addison-Wesley Professional, 2010.
10. M. Elaasar and L. Briand. An overview of uml consistency management. *Carleton University, Canada, Technical Report SCE-04-18*, 2004.
11. R. B. France, D. kyoo Kim, S. Ghosh, and E. Song. A uml-based pattern specification technique. *IEEE Transactions on Software Engineering*, pages 193–206, 2004.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP' 93 - Object-Oriented Programming*, Lecture Notes in Computer Science, pages 406–431. Springer Berlin Heidelberg, 1993.
13. D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. *Foundations of component-based systems*, pages 47–68, 2000.
14. N. B. Harrison and P. Avgeriou. Leveraging architecture patterns to satisfy quality attributes. In *Proceedings of the First European Conference on Software Architecture*, pages 263–270, 2007.
15. N. B. Harrison, P. Avgeriou, and U. Zdun. Using patterns to capture architectural decisions. *IEEE Softw.*, 24(4):38–45, July 2007.
16. ISO/IEC/IEEE 42010:2011. *Systems and Software Engineering - Architecture Description*. ISO, Geneva, Switzerland.
17. A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 109–120. IEEE Computer Society, 2005.
18. A. Jansen, J. van der Ven, P. Avgeriou, and D. Hammer. Tool support for architectural decisions. In *Software Architecture, 2007. WICSA '07. The Working IEEE/IFIP Conference on*, pages 4–4, 2007.
19. P. Könemann and O. Zimmermann. Linking design decisions to design models in model-based software development. In *Proceedings of the 4th European Conference on Software Architecture*, pages 246–262. Springer-Verlag, 2010.
20. P. Kruchten, P. Lago, and H. van Vliet. Building up and reasoning about architectural knowledge. In *Proceedings of the Second international conference on Quality of Software Architectures*, pages 43–58. Springer-Verlag, 2006.
21. M. Küster. Architecture-centric modeling of design decisions for validation and traceability. In *European Conference on Software Architecture (ECSA)*, pages 184–191, 2013.
22. F. J. Lucas, F. Molina, and A. Toval. A systematic review of uml model consistency management. *Information and Software Technology*, 51(12):1631 – 1645, 2009.

23. I. Lytra, H. Tran, and U. Zdun. Supporting consistency between architectural design decisions and component models through reusable architectural knowledge transformations. In *Proceedings of the 7th European Conference on Software Architecture*, ECSA'13. Springer-Verlag, 2013.
24. N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling software architectures in the unified modeling language. *ACM Trans. Softw. Eng. Methodol.*, pages 2–57, Jan. 2002.
25. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, pages 70–93, 2000.
26. P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*, pages 264–278. Springer-Verlag, 2005.
27. O.M.G. Model-driven architecture. `http://wwww.omg.org/mda`.
28. OMG. Object Constraint Language, OCL Version 2.0, formal/2006-05-01. Technical report, OMG, 2006.
29. F. Oquendo. Pi-adl: an architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes*, pages 1–14, May 2004.
30. J. Savolainen, J. Kuusela, T. Mnnist, and A. Nyyssnen. Experiences in making architectural decisions during the development of a new base station platform. In M. A. Babar and I. Gorton, editors, *ECSA*, Lecture Notes in Computer Science, pages 425–432. Springer, 2010.
31. M. Shahin, P. Liang, and M. Khayyambashi. Architectural design decision: Existing models and tools. In *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009*, pages 293–296, Sept 2009.
32. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Addison-Wesley Professional, 2008.
33. A. Tang, Y. Jin, and J. Han. A rationale-based architecture model for design traceability and reasoning. *Journal of Systems and Software*, pages 918–934, 2007.
34. M. T. T. That, S. Sadou, and F. Oquendo. Using architectural patterns to define architectural decisions. In *Joint IEEE/IFIP Working Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, pages 196–200, 2012.
35. M. T. T. That, S. Sadou, F. Oquendo, and I. Borne. Composition-centered architectural pattern description language. In *Proceedings of the 7th European Conference on Software Architecture*. Springer-Verlag, 2013.
36. E. Thomas. *SOA Design Patterns*. Prentice Hall, 2009.
37. C. Tibermacine, R. Fleurquin, and S. Sadou. Preserving architectural choices throughout the component-based software development process. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 121–130, 2005.
38. C. Tibermacine, R. Fleurquin, and S. Sadou. A family of languages for architecture constraint specification. *J. Syst. Softw.*, pages 815–831, 2010.
39. C. Tibermacine, S. Sadou, C. Dony, and L. Fabresse. Component-based specification of software architecture constraints. In *Proceedings of the 14th international ACM Sigsoft symposium on Component Based Software Engineering*, pages 31–40, 2011.
40. J. Tyree and A. Akerman. Architecture decisions: Demystifying architecture. *IEEE Software*, pages 19–27, 2005.
41. R. Wojcik, F. Bachmann, L. Bass, P. C. Clements, P. Merson, R. Nord, and W. G. Wood. Attribute-Driven Design (ADD), Version 2.0. Technical report, Software Engineering Institute, Nov. 2006.
42. A. Zalewski, S. Kijas, and D. Sokolowska. Capturing architecture evolution with maps of architectural decisions 2.0. In *Proceedings of the 5th European Conference on Software Architecture*, pages 83–96. Springer-Verlag, 2011.
43. U. Zdun, P. Avgeriou, C. Hentrich, and S. Dustdar. Architecting as decision making with patterns and primitives. In *Proceedings of the 3rd international workshop on Sharing and reusing architectural knowledge*, SHARK '08, pages 11–18, New York, NY, USA, 2008. ACM.
44. O. Zimmermann. Architectural decisions as reusable design assets. *Software, IEEE*, pages 64–69, 2011.

45. O. Zimmermann. Architectural decision identification in architectural patterns. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, WICSA/ECSA '12, pages 96–103, New York, NY, USA, 2012. ACM.
46. O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster. Reusable architectural decision models for enterprise application development. In *Proceedings of the Quality of Software Architectures 3rd international conference on Software architectures, components, and applications*, pages 15–32. Springer-Verlag, 2007.
47. O. Zimmermann, U. Zdun, T. Gschwind, and F. leymann. Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method. In *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pages 157–166. IEEE Computer Society, 2008.

Appendix A Formalized SOA pattern

Table 5 shows the list of formalized SOA patterns along with the number of architectural elements, the involved roles and the number of multiplicity elements.

Appendix B Formalized CBA pattern

Table 6 shows the list of formalized CBA patterns along with the number of architectural elements, the involved roles and the number of multiplicity elements.

Appendix C List of architectural models

Table 7 shows the list of architectural models used in the evaluation along with the applied patterns and their frequency.

Appendix D The SOA mapping meta-model

Figure 19 shows the mapping meta-model in case of SOA pattern. Elements from the left side of the figure are those from SCA meta-model. Elements from the right side of the figure are those from the SOA pattern meta-model. Each pair of these elements is linked by an mapping element from the mapping meta-model (in the middle of the figure) via the source and target references. All mapping elements are contained in the root mapping element.

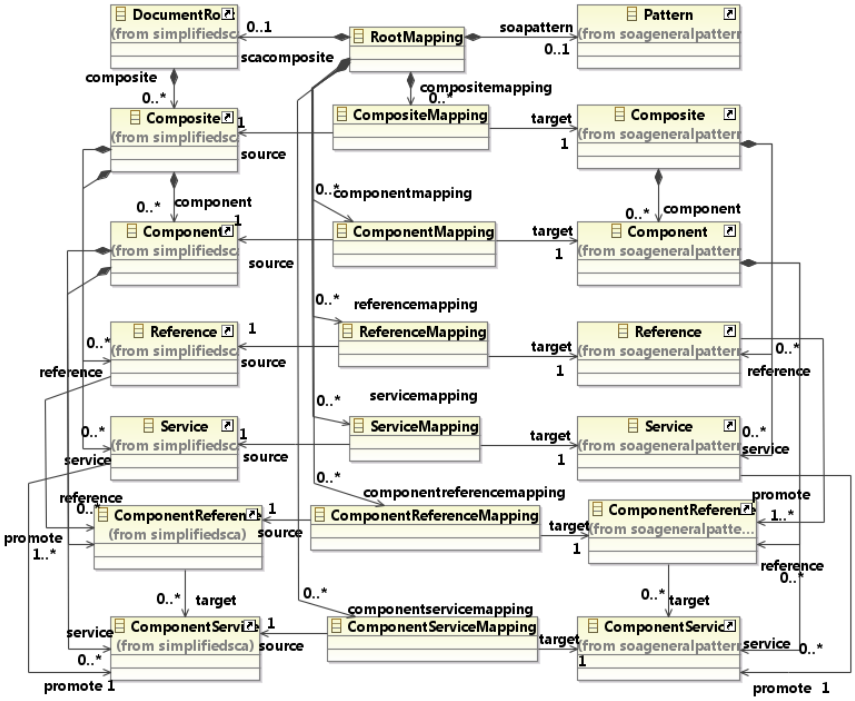

Fig. 19 SOA Mapping Meta-model

Table 5 List of formalized SOA patterns

Pattern category	Patterns	Nb of elements	Roles	Nb of multi-plicities
Service inventory	Rules Centralization	2	Service, Rule service	1
	Dual Protocols	1	Protocol	0
	Service Grid	1	Service	0
	Inventory Endpoint	3	Internal inventory service, Inventory endpoint, External consumer	2
	State Repository	2	Service, State repository	1
Service	Service Façade	3	Service, Façade, Consumer	0
	Service Data Replication	2	Service, Replicated database	1
	Partial State Deferral	2	Service, Deferral state repository	0
	Partial Validation	2	Service, Data validator	0
	Decoupled Contract	2	Service, Service contract repository	0
	Legacy Wrapper	2	Legacy Component, Wrapper Component	1
	Exception Shielding	2	Service, Exception Shield	0
	Message Screening	2	Service, Message screener	0
	Trusted Subsystem	1	Service, Trusted Subsystem	0
	Service Perimeter Guard	2	Internal service, Perimeter service	1
	Proxy Capability	2	Service, Proxy	0
	Decomposed Capacity	2	Service, Decomposed proxy	1
	Canonical Protocol	1	Service with uniform protocol	0
	Redundant Implementation	1	Redundant service	1
Service composition	Intermediate Routing	2	Service, Intermediate logic router	0
	Asynchronous Queuing	3	Service, Intermediary buffer, Consumer	0
	Brokered Authentication	3	Service, Broker, Consumer	1
	Data Format Transformation	3	Service, Intermediary data formatter, Legacy component	1
	Service Agent	1	Service Agent	0
	Agnostic Sub-controller	2	Service, Sub-controller	0

Table 6 List of formalized CBA patterns

Pattern category	Patterns	Nb of elements	Roles	Nb of multi-plicities
Layered	Layers	2	Layer, Layer connector	1
	Indirection Layer	4	Client layer, Indirection layer, Sub-system, Layer connector	0
Data flow	Pipes and Filters	2	Filter, Pipe	1
Data-centered	Shared Repository	3	Client, Repository, Data accessor	1
	Active Repository	3	Client, Active repository, Data accessor	1
	Blackboard	3	Blackboard, Knowledge source, Data accessor	1
Adaptation	Microkernel	5	Client, External server, Micro kernel, Internal server, Layer connector	2
	Interceptor	4	Client layer, Interceptor, Sub-system, Layer connector	0
User interaction	Model-View-Controller	4	Model, View, Controller, MVC connector	0
	Presentation-Abstraction-Control	2	PAC agent, PAC connector	0
	C2	2	Component, Connector	0
Component interaction	Client-Server	3	Client, Server, Request/Reply connector	1
	Peer to peer	2	Peer, Peer connector	1
Distribution	Broker	4	Client, Server, Broker, Broker connector	0

Table 7 List of architectural models

Architectural models	Description	Applied patterns	Frequency
BRM	A revenue management system	Layers	1
DPS	A digital publishing system	Repository	2
JITC	A source code comprehension aiding system	Pipes and Filters	1
		Repository	1
BTS	A bond trading system	Pipes and Filters	1
GCC	A digital TV system middle-ware	Pipes and Filters	1
JBoss	An open source J2EE implementation	Broker	1
		Microkernel	1
		Pipes and Filters	1
Vistrails	A data exploration and visualization open-source system	Pipes and Filters	1
		Repository	1
CoCoME	A supermarket sales system	Layers	1